AFRL-RI-RS-TR-2009-41
Final Technical Report
February 2009

EVOLUTION OF THE ETHANE ARCHITECTURE

Nicira Networks, Inc.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-41 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
JAMES L. SIDORAN Work Unit Manager	WARREN H. DEBANY, Jr. Technical Advisor Information Grid Division Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEB 2009	Final	Dec 07 – Sep 08

4. TITLE AND SUBTITLE

EVOLUTION OF THE ETHANE ARCHITECTURE

5a. CONTRACT NUMBER
FA8750-08-C-0023

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Martin Casado and Scott Shenker

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
17

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Nicira Networks, Inc.
385 Sherman Ave, Suite 14
Palo Alto, CA 94306-2639

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIGA
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-41

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-0335

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Ethane architecture, developed at Stanford University, demonstrated that a novel approach to building secure networks could support superior low-level security and flexible policy-based control over individual flows. However, Ethane only provided operators with a single function: policy-based access control. Moreover, Ethane's policy was expressed in a language that did not have a rigorous logical foundation. Almost a year of subsequent work, reported on here, extended Ethane to address these two shortcomings. First, the Ethane architecture was evolved from Ethane's narrowly targeted design to a fully general network operating system called NOX, which provides users with full-blown programmatic interface. Second, the policy language has evolved from the Ethane's primitive pol-eth to a much more powerful and rigorously analyzed Flow-Based Security Language (FSL). This report describes these two advances.

15. SUBJECT TERMS

Secure policy architecture; enterprise network security; network management; and policy enforcement

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James L. Sidoran
U	U	U	UU	42	19b. TELEPHONE NUMBER (*Include area code*) N/A

Contents

List of Figures

List of Tables

Chapter 1

Summary

The Ethane architecture, developed at Stanford University, demonstrated that a novel approach to building secure networks could support robust low-level security and flexible policy-based control over individual flows. However, Ethane only provided operators with a single function: policy-based access control. Moreover, Ethane's policy was expressed in a language that did not have a rigorous logical foundation. Almost a year of subsequent work, reported on here, extended Ethane to address these two shortcomings. First, the Ethane architecture was evolved from Ethane's narrowly targeted design to a fully general *network operating system* called NOX, which provides users with powerful programmatic interface. Second, the policy language has evolved from the Ethane's primitive *pol-eth* to a much more powerful and rigorously analyzed *Flow-Based Security Language* (FSL). This report describes these two advances.

Chapter 2

Introduction

Enterprise networks are often large, run a wide variety of applications and protocols, and typically operate under strict reliability and security constraints; thus, they represent a challenging environment for network management. The stakes are high, as enterprise productivity and security can be severely hampered by network misconfigurations or break-ins. Yet the current solutions are weak, making enterprise network management both expensive and error-prone. Indeed, most networks today require substantial manual configuration by trained operators [17, 50, 58, 55] to achieve even moderate security [56]. A Yankee Group report found that 62% of network downtime in multi-vendor networks comes from human-error and that 80% of IT budgets is spent on maintenance and operations [38].

There have been many attempts to make networks more manageable and more secure. One approach introduces proprietary middleboxes that can exert their control effectively only if placed at network choke-points. If traffic accidentally flows (or is maliciously diverted) around the middlebox, the network is no longer managed nor secure [58]. Another approach is to add functionality to existing networks—to provide tools for diagnosis; to offer controls for VLANs, access-control lists, and filters to isolate users; to instrument the routing and spanning tree algorithms to support better connectivity management; and then to collect packet traces to allow auditing. This can be done by adding a new layer of protocols, scripts, and applications [1, 18] that help automate configuration management in order to reduce the risk of errors. However, these solutions hide the complexity, not reduce it. Moreover, they have to be constantly maintained to support the rapidly changing and often proprietary management interfaces exported by the managed elements.

Rather than building a new layer of complexity on top of the network, the Ethane project explored the question: *How could we change the enterprise network architecture to make it more manageable?* As a result of our investigations, we decided to build Ethane around three fundamental principles that we feel are important to any network management solution:

The network should be governed by policies declared over high-level names. Networks are most easily managed in terms of the entities we seek to control—such as users, hosts, and access points—rather than in terms of low-level and often dynamically-allocated addresses. For example, it is convenient to declare which services a user is allowed to use and to which machines they can connect.

Network routing should be policy-aware. Network policies dictate the nature of connectivity between communicating entities and therefore naturally affect the paths that packets take. This is in contrast to today's networks in which forwarding and filtering use different mechanisms rather than a single integrated approach.

A policy might require packets to pass through an intermediate middlebox; for example, a guest user might be required to communicate via a proxy, or the user of an unpatched operating system might be required to communicate via an intrusion-detection system. Policy may also specify service priorities for different classes of traffic. Traffic can receive more appropriate service if its path is controlled; directing real-time communications over lightly loaded paths, important communications over redundant paths, and private communications over paths inside a trusted boundary would all lead to better service.

The network should enforce a strong binding between a packet and its origin. Today, it

is notoriously difficult to reliably determine the origin of a packet: Addresses are dynamic and change frequently, and they are easily spoofed. The loose binding between users and their traffic is a constant target for attacks in enterprise networks. If the network is to be governed by a policy declared over high-level names (*e.g.*, users and hosts) then packets should be identifiable, without doubt, as coming from a particular physical entity. This requires a strong binding between a user, the machine they are using, and the addresses in the packets they generate. This binding must be kept consistent at all times, by tracking users and machines as they move.

To achieve these aims, the Ethane project followed the lead of the 4D project [29] and adopted a centralized control architecture. Centralized solutions are normally an anathema for networking researchers, but we feel it is the proper approach for enterprise management. IP's best-effort service model is both simple and unchanging, well-suited for distributed algorithms. Network management is quite the opposite; its requirements are complex and variable, making it quite hard to compute in a distributed manner.

There are many standard objections to centralized approaches, such as resilience and scalability. However, our experience with Ethane suggests that standard replication techniques can provide excellent resilience, and current CPU speeds make it possible to manage all control functions on a sizable network (*e.g.*, 25,000 hosts) from a single commodity PC.

Ethane bears substantial resemblance to SANE, our previously-proposed clean-slate approach to enterprise security [21]. SANE was, as are many clean-slate designs, difficult to deploy and largely untested. While SANE contained many valuable insights, Ethane extends this previous work in two main ways:

- *Security follows management*: Enterprise security is, in many ways, a subset of network management. Both require a network policy, the ability to control connectivity, and the means to observe network traffic. Network management wants these features so as to control and isolate resources, and then to diagnose and fix errors, whereas network security seeks to control who is allowed to talk to whom, and then to catch bad behavior before it propagates. When designing Ethane, we decided that a broad approach to network management would also work well for network security.

- *Incremental deployability*: SANE required a "fork-lift" replacement of an enterprise's entire networking infrastructure and changes to all the end-hosts. While this might be suitable in some cases, it is clearly a significant impediment to widespread adoption. Ethane is designed so that it can be incrementally deployed within an enterprise: it does not require any host modifications, and Ethane Switches can be incrementally deployed alongside existing Ethernet switches.

The technical details of the current Ethane design can be found at [22]. Thus, the purpose of the present report is not review the Ethane design, but to discuss extensions to this design. Our experiences with Ethane pointed to two main areas where Ethane was lacking:

- *Generality*: Ethane was designed to provide a particular level of policy-based control. However, there are many other network management tasks that don't fit into this category. Thus, we decided to generalize from Ethane's single focus into a fully-general network operating system, called NOX. NOX provides a general programmatic interface to the network, allowing network operators a fully flexible tool for managing their network.

- *Policy Language*: Ethane used a policy called *pol-eth* for policy declarations. While we did not encounter any practical problems with this language, it was not designed with any rigorous foundations. We have designed a follow-on language, FSL, that is far more rigorous and addresses important issues in large-scale networks with distributed administrative control.

This report focuses on these two topics. They are two distinc areas of interest, so they are treated in separate sections in the following chapter.

Chapter 3

Methods, Assumptions, and Procedures

3.1 From Ethane to NOX

3.1.1 Introduction

As anyone who has operated a large network can attest, enterprise networks are difficult to manage. That they have remained so despite significant commercial and academic efforts suggests the need for a different network management paradigm. Here we turn to operating systems as an instructive example in taming management complexity.

In the early days of computing, programs were written in machine languages that had no common abstractions for the underlying physical resources. This made programs hard to write, port, reason about, and debug. Modern operating systems facilitate program development by providing controlled access to high-level abstractions for resources (*e.g.*, memory, storage, communication) and information (*e.g.*, files, directories). These abstractions enable programs to carry out complicated tasks safely and efficiently on a wide variety of computing hardware.

In contrast, networks are managed through low-level configuration of individual components. Moreover, these configurations often depend on the underlying network; for example, blocking a user's access with an ACL entry requires knowing the user's current IP address. More complicated tasks require more extensive network knowledge; forcing guest users' port 80 traffic to traverse an HTTP proxy requires knowing the current network topology and the location of each guest. In this way, an enterprise network resembles a computer without an operating system, with network-dependent component configuration playing the role of hardware-dependent machine-language programming.

What we clearly need is an "operating system" for networks, one that provides a uniform and centralized programmatic interface to the entire network.[1] Analogous to the read and write access to various resources provided by computer operating systems, a network operating system provides the ability to *observe* and *control* a network.

A network operating system does not manage the network itself; it merely provides a programmatic interface. *Applications* implemented on top of the network operating system perform the actual management tasks.[2] The programmatic interface should be general enough to support a broad spectrum of network management applications.

[1] In the past, the term *network operating system* referred to operating systems that incorporated networking (*e.g.*, Novell NetWare), but this usage is now obsolete. We are resurrecting the term to denote systems that provide an execution environment for programmatic control of the network.

[2] In the rest of this report, the term applications will refer exclusively to management programs running on a network operating system.

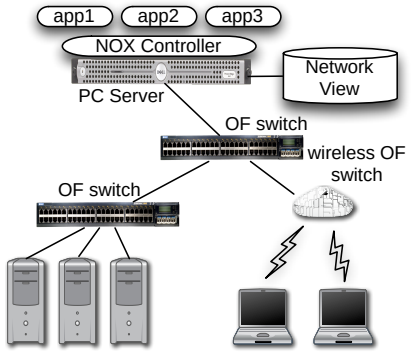

Figure 3.1: Components of a NOX-based network: OpenFlow (OF) switches, a server running a NOX controller process and a database containing the network view.

Such a network operating system represents two major conceptual departures from the status quo. First, the network operating system presents programs with a *centralized* programming model[3]; programs are written as if the entire network were present on a single machine (*i.e.*, one would use Dijkstra to compute shortest paths, not Bellman-Ford). This requires (as in [15, 29, 47] and elsewhere) centralizing network state. Second, programs are written in terms of high-level abstractions (*e.g.*, user and host names), not low-level configuration parameters (*e.g.*, IP and MAC addresses). This allows management directives to be enforced independent of the underlying network topology, but it requires that the network operating system carefully maintain the bindings (*i.e.*, mappings) between these abstractions and the low-level configurations.

Thus, a network operating system allows management applications to be written as centralized programs over high-level names as opposed to the distributed algorithms over low-level addresses we are forced to use today. While clearly a desirable goal, achieving this transformation from distributed algorithms to centralized programming presents significant technical challenges, and the question we pose here is: *Can one build a network operating system at significant scale?* We argue for an affirmative answer to this question via proof-by-example; herein we describe a network operating system called NOX (freely available at http://www.noxrepo.org) that achieves the goals outlined above.

3.1.2 NOX Overview

We now give an overview of NOX by discussing its constituent components, observation and control granularity, switch abstraction, basic operation, scaling, status and public release.

Components Figure 3.1 shows the primary components of a NOX-based network: a set of switches and one or more network-attached servers. The NOX software (and the management applications that run on NOX) run on these servers. The NOX software can be thought of as involving several different *controller* processes (typically one on each network-attached server) and a single *network view* (this is kept in a database running on one of the servers).[4] The network view contains the results of NOX's network observations; applications use this state to make management decisions. For NOX to control network traffic, it must manipulate network switches; for this purpose we have chosen to use switches that support the OpenFlow (OF) switch abstraction [4, 45], which we describe later in this section.

[3]By *centralized* we allude to a shared memory programming model. However, as we discuss in Section 3.1.3, different memory locations may have different access overheads.

[4]For resilience, this database can be replicated, but these replicas must be kept consistent (as can be done using traditional replicated database techniques).

Granularity An early and important design issue was the granularity at which NOX would provide observation and control. Choosing the granularity involves trading off scalability against flexibility, and both are crucial for managing large enterprise networks with diverse requirements. For observation, NOX's network view includes the switch-level topology; the locations of users, hosts, middleboxes, and other network elements; and the services (*e.g.*, HTTP or NFS) being offered. The view includes all bindings between names and addresses, but does *not* include the current state of network traffic. This choice of observation granularity provides adequate information for many network management tasks and changes slowly enough that it can be scalably maintained in large networks.

The question of control granularity was more vexing. A centralized per-packet control interface would clearly be infeasible to implement across any sizable network. At the other extreme, operating at the granularity of prefix-based routing tables would not allow sufficient control, since all packets between two hosts would have to follow the same path. For NOX we chose an intermediate granularity: *flows* (similar in spirit to [46]). That is, once control is exerted on some packet, subsequent packets with the same header are treated in the same way. With this flow-based granularity, we were able to build a system that can scale to large networks while still providing flexible control.

Switch Abstraction Management applications control network traffic by passing instructions to switches. These switch instructions should be independent of the particular switch hardware, and should support the flow-level control granularity described above. To meet these requirements, NOX has adopted the OpenFlow switch abstraction (see [4, 45] for details). In OpenFlow, switches are represented by flow tables with entries of the form:[5]

$$\langle header : counters, actions \rangle$$

For each packet matching the specified header, the counters are updated and the appropriate actions taken. If a packet matches multiple flow entries, the entry with the highest priority is chosen. An entry's header fields can contain values or ANYs, providing a TCAM-like match to flows. The basic set of OpenFlow actions are: forward as default (*i.e.*, forward as if NOX were not present), forward out specified interface, deny, forward to a controller process, and modify various packet header fields (*e.g.*, VLAN tags, source and destination IP address and port). Additional actions may later be added to the OpenFlow specification.

Operation When an incoming packet matches a flow entry at a switch, the switch updates the appropriate counters and applies the corresponding actions. If the packet does not match a flow entry, it is forwarded to a controller process.[6] These unmatching packets often are the first packet of a flow (hereafter, flow-initiations); however, the controller processes may choose to receive all packets from certain protocols (*e.g.*, DNS) and thus will never insert a flow entry for them. NOX applications use these flow-initiations and other forwarded traffic to *(i)* construct the network view (observation) and *(ii)* determine whether to forward traffic, and, if so, along which route (control).

As an example of *(i)*, we have built applications that use DNS, DHCP, LLDP, and flow-initiations to construct the network view, including both the network topology and the set of name-address bindings. We have also built applications that intercept authentication traffic to perform user and host authentications (using, for example, 802.1x). As an example of *(ii)*, we have developed access-control and routing applications that determine if a flow should be allowed, compute an appropriate L2 route, install flow entries in all the switches along the path, and then return the packet to the originating switch (which then forwards it along the designated path).

Scaling Our confidence in the scalability of NOX follows from considering the spectrum of timescales and consistency requirements. In terms of timescales, NOX processing occurs at three very different rates:

[5]It is important to distinguish between the levels of abstraction provided by OpenFlow and NOX. NOX provides network-wide abstractions, much like operating systems provide system-wide abstractions. OpenFlow provides an abstraction for a particular network component, and is thus more analogous to a device driver.

[6]Typically, only the first 200 bytes of the first packet (including the header) are forwarded to the controller, but the controller may adjust this, or request additional packets be forwarded, if more information is deemed necessary.

- Packet arrivals: this is on the order of millions of arrivals per second for a 10Gbps link.
- Flow initiations: with NOX's definition of flow (which is typically more persistent than NetFlow's definition), the flow-initiation rate is typically one or more orders of magnitude less than the packet arrival rate.
- Changes in the network view: in our deployment experience this is on the order of tens of events per second for networks of thousands of hosts.

In terms of consistency, the only network state that is global (*i.e.*, must be used consistently across the controller processes) is the network view; this consistency requirement arises because applications use data from the network view to make control decisions, and those decisions should be the same no matter to which controller instance the flow has been sent. In contrast, since neither packet state nor flow state are part of the network view, they can be kept in local storage (*i.e.*, packet state in switches, and flow state in controller instances).

Thus, for the categories of events that occur on rapid timescales, NOX can use parallelism; packet arrivals are handled by individual switches without global per-packet coordination, and flow-initiations are handled by individual controller instances without global per-flow coordination. Flows can be sent to any controller instance, so the capacity of the system can be increased by adding more servers running controller processes.

The one global data structure, the network view, changes slowly enough that it can be maintained centrally (or, for resilience, it can be kept consistently on a small set of replicas) for even very large networks.

To give some rough numbers, a single controller process running on a generic PC can currently handle 100,000 flow initiations per second, more than sufficient for the large campus networks we've measured in previous work [22].

Implementation Status We have been developing NOX for over a year and have been running it in our internal network of roughly 30 hosts for over 6 months. It is our only means of network connectivity. NOX runs in user-space on the network servers. Applications are written in either Python or C++ and are loaded dynamically. The core infrastructure and speed-critical functions of NOX are implemented in C++ (currently about 32,000 lines). The network view is a set of indexed hashtables, with extensions for distributed access with local caching to aid scaling across multiple controller instances.

Public Release Our brief presentation of NOX is only the first step in our "proof" that one can build a network operating system. NOX is freely available at http://www.noxrepo.org (GPL license), and we invite the community to build additional applications on NOX. The community's experience will provide the definitive answer to the question of whether NOX provides a useful abstraction for network management.

3.1.3 Programmatic Interface

NOX's programmatic interface is conceptually quite simple, revolving around events, a namespace, and the network view.

Events Enterprise networks are not static: flows arrive and depart, users come and go, and links go up and down. To cope with these change events, NOX applications use a set of handlers that are registered to execute when a particular event happens. Event handlers are executed in order of their priority (which is specified during handler registration). A handler's return value indicates to NOX whether to stop execution of this event, or to continue by passing the event to the next registered handler.

Some events are generated directly by OpenFlow messages, such as *switch join*, *switch leave*, *packet received*, and *switch statistics received*. Other events are generated by NOX applications as a result of processing these low-level events and/or other application-generated events. For example, NOX includes applications that will authenticate a user by redirecting raw HTTP traffic (a *packet received* event) to a

captive web portal. Once the user is authenticated, the NOX application generates a *user authenticated* event which can be used by other applications.

We have implemented applications that reconstruct the switch and host level topology, discover network services, authenticate users and hosts, enforce network policy, and detect network scanning (to name a few). Each of these services is coupled with an associated event that can be leveraged by other applications.

Network View and Namespace NOX includes a number of "base" applications that construct the network view and maintain a high-level namespace that can be used by other applications. These applications handle user and host authentication, and infer host names by monitoring DNS. The inclusion of high-level names and their bindings in the network view allows any application to convert a high-level name into low-level addresses (or vice versa), allowing applications to be written in a topology independent manner. To perform such conversions, high-level declarations can be "compiled" against the network view to produce low-level lookup functions that are enforced per-packet. These functions are recompiled on each change to the network view. In Section 3.1.4 we describe how this is used in practice.

Because the network view must be consistent and made available to all NOX controller instances, writing to it incurs some expense. Thus, NOX applications should only write to it when a *change* is detected in the network, and not for every received packet. This is similar to the access model provided by NUMA memory architectures. Also like NUMA, the worst result of a poorly written application is performance degradation, not incorrect function.

Control Management applications exert network control through OpenFlow. The OpenFlow switch abstraction allows applications to insert entries, delete entries, or read counters from entries in the flow table. Through its ability to modify these flow table entries, a management application has complete control of L2 routing, packet header manipulation, and ACLs. With the definition of additional switch actions, applications could also control common per-packet processing primitives such as encryption and rate-limiting.

The OpenFlow abstraction is intended to be general, so it can only require features common to most switches. Therefore we do not expect that the actions standardized by the OpenFlow Switch Consortium will include custom per-packet processing (*e.g.*, deep packet inspection). However, NOX applications can direct traffic through specialized middleboxes, so NOX-managed networks can still take advantage of the latest per-packet processing technology.

Higher-Level Services NOX includes a set of "system libraries" to provide efficient implementations of functions common to many network applications. These include a routing module, fast packet classification, standard network services (such as DHCP and DNS), and a policy-based network filtering module.

Interface and Runtime Limitations Our goal with NOX is to build a practical platform for writing network management applications that can scale to large networks. So far, we have focused on scalability and functionality, and have yet to address a number of practical considerations that would improve the safety and isolation of NOX applications. For example, we assume that there is coordination between application writers and do not try to protect against malicious or faulty applications: a bad application can drop an event, overwrite random memory, or hang the system with an infinite loop. We feel providing inter-application coordination and isolation is a rich area for exploration in which the Maestro project has already made progress [16].

3.1.4 Example Applications

We now describe a few examples of NOX applications. To illustrate NOX's programming model, we start with two oversimplified examples. To give an idea of NOX's power, we then describe how we used NOX to re-implement Ethane [22], an identity-based access-control system. To convey NOX's flexibility, we end with two ongoing projects aiming for novel network functionality.

```
# On user authentication, statically setup VLAN tagging
# rules at the user's first hop switch
def setup_user_vlan(dp, user, port, host):
    vlanid = user_to_vlan_function(user)
    # For packets from the user, add a VLAN tag
    attr_out[IN_PORT] = port
    attr_out[DL_SRC] = nox.reverse_resolve(host).mac
    action_out = [(nox.OUTPUT, (0, nox.FLOOD)),
                  (nox.ADD_VLAN, (vlanid))]
    install_datapath_flow(dp, attr_out, action_out)
    # For packets to the user with the VLAN tag, remove it
    attr_in[DL_DST] = nox.reverse_resolve(host).mac
    attr_in[DL_VLAN] = vlanid
    action_in = [(nox.OUTPUT, (0, nox.FLOOD)),
                 (nox.DEL_VLAN)]
    install_datapath_flow(dp, attr_in, action_in)
nox.register_for_user_authentication(setup_user_vlan)
```

Figure 3.2: An example NOX application written in Python that statically sets VLAN tagging rules on user authentication. A complete application would also add VLAN removal rules at all end-point switches.

```
scans = defaultdict(dict)
def check_for_scans(dp, inport, packet):
    dstid = nox.resolve_host_dest(packet)
    if dstid == None:
        scans[packet.l2.srcaddr][packet.l2.dstaddr] = 1
        if packet.l3 != None:
            scans[packet.l2.srcaddr][packet.l3.dstaddr] = 1
        if len(scans[packet.l2.srcaddr].keys()) > THRESHOLD:
            print nox.resolve_user_source_name(packet)
            print nox.resolve_host_source_name(packet)
# To be called on all packet-in events
nox.register_for_packet_in(check_for_scans)
```

Figure 3.3: A simplistic NOX application written in Python that attempts to detect scanning hosts by tracking the number of unique, unknown L2 and L3 destinations attempted by a single host.

Two Simple Examples

User-based VLAN Tagging Figure 3.2 contains a simplistic NOX application that sets up VLAN tagging rules on user authentication based on a predefined user-to-VLAN mapping. NOX is responsible for detecting all flow-initiations, attributing the flow to the correct user, host and ingress access point, and dispatching the event to the application. This would provide attribution in logging and diagnostics; it could also, with minor modifications to the routing module, support traffic isolation.

Simplistic Scan Detection The application in Figure 3.3 attempts to detect scanning hosts by counting the number of unique L2 and L3 destinations a host tries to contact that have not authenticated. NOX has access to traffic across the network, and it can leverage the network view which tracks all authenticated hosts on the network. Contrast this simple implementation to that in [37], where scan detection required a traffic choke-point and heuristics to guess whether an IP address is active.

Ethane

We recently built a system called Ethane that provides network-wide access-control using a centralized declaration of policy over high-level principals (*i.e.*, entities in the network view namespace) [22]. Because we have implemented Ethane both with and without NOX, Ethane is an instructive example of how NOX simplifies

management application development: our stand-alone Ethane implementation required over 45,000 lines of C++, while our implementation within NOX required a few thousand lines of Python.

Ethane has two requirements that make it difficult to implement using traditional network management techniques: *(i)* it requires knowledge of the principals on the network (*e.g.*, users, nodes[7]), and *(ii)* it requires control over routing at the granularity of a flow's 7-tuple (source user, source host, first-hop switch, protocol, destination user, destination host, last-hop switch).[8] Both of these are natively supported by NOX: the application has access to the source and destination principals associated with each event, and the routing module supports route computation with constraints.

Implementing the basic Ethane functionality within NOX involves first checking each flow directly against the declared policy and then passing the resulting contraints to NOX's routing module. The only subtle aspect of the implementation is how to efficiently check flows against the policy, since a linear scan of the policy declaration file for each flow scales poorly as policy complexity increases. To improve performance in the average case, we dynamically construct efficient lookup trees from the policy declarations.

Other Applications

The following are two examples of areas where NOX is being employed to explore new functionality.

Power Management There has been significant recent interest in managing networks to save power [6, 31]. The two techniques most commonly discussed are *(i)* reducing the speed of underutilized links, or turning them off altogether, and *(ii)* providing proxies to intercept network *chatter* (only allowing necessary packets to reach hosts would make wake-on-LAN more effective). NOX's global view of the network and the routes currently in use facilitates the former technique, while NOX's interposition on all flow-initiations facilitates the latter. Thus, NOX is an ideal platform for implementing these techniques, and there is ongoing work [33] pursuing this opportunity.[9]

Home Networking Calvert *et al.*, in [19], trace many of the difficulties in managing home networks to the end-to-end nature of the Internet architecture, and propose that a more centralized network design be used in homes. NOX could centralize the observation and control functions of the home network, while preserving the decentralized nature of the datapath (thereby avoiding potential bottlenecks). NOX would also provide a natural platform on which one could build management tools to handle the many higher-layer configuration issues (directing machines to local printers, etc.) that bedevil home networks. There are two ongoing efforts exploring NOX's use in the home.

3.1.5 Related Work

The idea of giving control mechanisms a global view of the network was first developed in the context of the 4D project (see [15, 29, 47]). Providing this view required a new networking paradigm based on simple switches enslaved to a logically centralized decision element that oversees the full network. This centralized paradigm is more flexible, since new functionality can be centrally programmed at the decision element rather than requiring a new distributed algorithm, but raises the specter of a single point of failure. However, adequate resilience can be achieved by applying standard replication techniques to the decision element. Note that these replication techniques are completely decoupled from the network control algorithms, so they do not impede application innovation.

The goal of 4D systems is to control forwarding (*e.g.*, FIBs in routers), and thus their network view only includes the network infrastructure (*e.g.*, links, switches/routers). The SANE [23] and Ethane [22] projects provided a broader class of functionality by including a namespace for users and nodes in their network view and keeping track of the bindings between these names and the low-level MAC and IP addresses. SANE and

[7] We use the term node to refer to hosts, switches, and other network elements.

[8] This requirement arises from policies that impose routing constraints on a particular class of flows, such as requiring them to traverse specified middleboxes.

[9] We are indebted to Brandon Heller for pointing out NOX's potential role in power management.

Ethane also capture flow-initiation events, to exercise control at a finer granularity (per-flow control rather than FIB-based control).

NOX extends the SANE/Ethane work in two dimensions. First, it attempts to scale this centralized paradigm to very large systems. This scaling is made possible by the differing timescales discussed earlier. The second extension is allowing general programmatic control of the network. The SANE/Ethane systems were designed around a single application: identity-based access control. NOX aims to provide a general programming interface that makes it *easier* to support current management tasks and *possible* to provide more advanced management functionality. We have described a few example applications in this report, but only experience will reveal how generally useful this interface is. By making NOX freely available, we hope that the community will provide valuable feedback on NOX's utility.

A related project of particular note is Maestro [16] (developed in parallel to NOX), which is also billed as a "network operating system". In general, operating systems can be seen as revolving around two basic purposes: *(i)* providing applications with a higher level of abstraction so they need not deal with low-level details, and *(ii)* controlling the interactions between applications. NOX focuses on the first, while Maestro focuses on the second, "orchestrating" the control decisions made by various management applications. We think these approaches could be combined, and we hope to soon explore this possibility.

Given that industry has substantially more experience with practical enterprise management and security than academia, we would be remiss in not mentioning commercial solutions. Many commercial enterprise security products, such as firewalls, intrusion detection and protection systems, network mappers, and proxies, are network *appliances* in that a particular functionality is provided by an element (or several elements) placed in the network. This appliance approach is easy to deploy, but leads to a fragmented architecture in which the different appliances, and their functionality, are completely decoupled. Moreover, none of these appliances provide the flexibility of a general programming environment for network observation and control.

One area where these commercial products are far more advanced than NOX is their ability to deal with packet payloads. Many commercial solutions use deep-packet-inspection, proxies, and/or data-logging, while NOX generally only inspects the first few hundred bytes in a flow. However, NOX-based management applications can incorporate these payload processing technologies by directing flows through the appropriate middleboxes. Thus, one shouldn't view NOX as a replacement for current network management techniques, but instead as a framework that can coordinate and manage these ever-advancing technologies.

3.2 The FSL Policy Language

One of the key components of NOX is the policy language that allows operators to express their policy desires. In the following section, we describe this policy language; however, because we think it has applicability outside of NOX, we place our discussion in a more general context.

3.2.1 Introduction

A wide range of modern enterprises rely on their computer networks for day-to-day operations, and rely on their network administrators to configure security components so that the network is not misused by errant individuals or malware. Traditionally, network security policies have been enforced by manual configuration of individual network components such as router ACLs, firewalls, NATs and VLANs. However, emerging enterprise network designs and products, like NOX, increasingly support global policies that can be expressed using high level abstractions. While different network systems may employ different network-level mechanisms, modern designs now allow us to usefully separate policy specification from policy implementation, viewing the implementation mechanism as"compiling" high-level policy statements into the appropriate low-level configurations needed by a network's constituent components. In this setting, a central question that emerges is: what should the policy language for a large-scale high-speed operational network look like?

In this report, we propose a Flow-based Security Language (FSL) that is designed to provide meaningful, intuitive high-level concepts to network administrators, while at the same time supporting efficient implementation of the resulting policies in modern networks. Our language FSL is based on a restricted form of

DATALOG [36, 10, 41, 54, 28, 42, 9], which is widely used in connection with database systems and certain forms of logic programming, enhanced with a carefully-chosen use of negation [48, 10, 36]. As explained further in the body of this report, the specific design of FSL was guided by three practical issues:

General constraints: Many treatments of access control only consider the binary actions of *allow* and *deny*; however, modern network security requires a more general approach. Operators would like to impose connectivity constraints on the rate, the path (requiring it to either avoid or include certain nodes), and the directionality (such as restricting clients to outbound flows only) of communications.

Distributed and incremental authorship: There are many policy authors in large enterprises, and thus the policy language must cope gracefully with any conflicts between their policies. In addition, operators may need to quickly inject new policies into a system without ensuring the new entries are consistent with all currently-installed policies. Thus, FSL must gracefully and predictably handle two types of policy conflicts: those between administrators, and those within an administrator's own rule set.

Speed: Policy decisions must be enforced on each packet, and on network links this can easily reach millions of enforcement decisions per second.

While there is a large literature on general access policy languages (see, for example, [36, 14, 13, 5]), prior languages are generally geared toward general computing resources or operating system concepts such as user names, host names, groups, services, and so on. As a result, most theoretically-grounded policy languages have never been tested against the gritty realities of operational networks. In particular, the need to process policies at line-speed and the challenges facing operators to declare fully consistent policies, have rarely, if ever, been addressed in previous policy language designs.

3.2.2 Design Requirements

FSL is designed to be practically useful. This requires policies be writable by real operators and enforceable by real networks. Thus, FSL's design constraints come from the requirements of *writing* and *implementing* policies. In this section we first discuss these two requirements in general.

Writing Policies

Because it is intended for use by real operators, FSL should make it simple to declare and maintain policies. As time goes on, we expect organizations to accumulate a large number of policies. When an operator is faced with a situation requiring a change in policy, it would be unreasonable to require the operator to modify the entire policy corpus to ensure a newly written (or rewritten) policy has no conflicts with any other policies. Instead, we want to make it possible for operators to incrementally add policy statements as new requirements arise, even if these new statements conflict with older policy statements.

In addition, most large enterprise networks have several systems administrators, each with their own (sometimes overlapping) domain of control. The language needs to accommodate such distributed authorship of policies, with its inevitable conflicts.

Thus, FSL must have a clean model for reconciling conflicts, both within policies written by a single operator, and between policies written by different operators.

Implementing Policies

Policy decisions must be enforced on each packet, which places a very stringent performance requirement on any policy system. We can relax this requirement by assuming that policies are "flow-based." Operationally, by flow-based we mean that if two packets have the same header, then the policy dictates the same actions for the two packets. In a flow-based system, each time a policy decision is made, the packet header and the resulting policy action is (logically) inserted into a table of $\langle header; action \rangle$ entries, with entries timing out after a period of inactivity. When a new packet arrives, the system first checks for a matching entry

in the flow table. If found, the system executes the corresponding action, otherwise it must make a policy decision. The performance requirement of the system is thus measured in terms of the number of new flows per second.

In our experience, large networks (on the order of 10,000 hosts) typically generate fewer than 10^4 flows per second. This is a demanding requirement, but is far smaller than that of a packet-based system where each packet requires an independent policy decision. For instance, a fully loaded 10 Gbps link has over 10^6 packet arrivals per second, with an average packet size of 400 bytes.

Policies dictate the actions that should be taken for a particular set of flows. The actions that operators want to implement include more than just *allow* or *deny*. They want to *rate-limit* flows, and force others to traverse *waypoints* such as proxies or firewalls. Another common policy is to protect clients from inbound connections using NAT, while allowing traffic to public servers within the same network via a DMZ.

Thus, the policy system must be able to deal with a set of various policy actions. As we explain later, the only requirement we place on these actions is that they are partially-ordered with respect to security (e.g., deny is more secure than allow, rate-limited is more secure than not limited, etc.). We say an outcome is consistent with a policy if it is at least as secure as the action dictated by the policy. Thus, a policy's action can be seen as a constraint on that flow, restricting the possible set of outcomes.

Formal Model

We can formalize the policy system as follows. The fundamental unit upon which the network acts is a *unidirectional flow*[10] which consists of values for the eight fields.

Definition 1 (Unidirectional Flow)**.** *A unidirectional flow (uniflow for short) is characterized by an eight-tuple.*

$$\langle u_{src}, h_{src}, a_{src}, u_{tgt}, h_{tgt}, a_{tgt}, prot, request \rangle$$

- $u_{src}, u_{tgt} \in U$ *(the set of users)*
- $h_{src}, h_{tgt} \in H$ *(the set of hosts)*
- $a_{src}, a_{tgt} \in A$ *(the set of access points)*
- $prot \in P$ *(the set of protocols)*
- $request \in \{true, false\}$

Uniflows constitute the input to the access control decision maker. A security policy for NOX associates every possible uniflow with a set of constraints, and for the remainder of the report, we suppose that a uniflow can be allowed, denied, waypointed (the route the uniflow takes through the network must include the stipulated hosts), forbidden to pass through certain waypoints (the route for the uniflow cannot include the stipulated hosts), and rate-limited. Formally, NOX policies are functions on uniflows as defined below, and FSL is a language for describing such functions.

Definition 2 (NOX Policy)**.** *A NOX policy is a function from unidirectional flows to two sets of hosts and a natural number. The first set of hosts corresponds to the set of required waypoints, the second set to the set of forbidden waypoints, and the natural number corresponds to the allowed communication rate limit in Mb/s.*

$$U \times H \times A \times U \times H \times A \times P \times \{T, F\} \rightarrow 2^H \times 2^H \times \mathcal{N}$$

[10]The term *unidirectional flow* is more appropriate than simply *flow* because the latter is often analogous to a nonce, which consists of two messages: one sent from the source to the destination and another one sent back. A unidirectional flow corresponds to exactly one of those messages.

3.2.3 FSL

Conceptually, FSL is a language for representing NOX Policies, policies that mandate which constraints should be applied to which unidirectional flows. It was designed so that given a uniflow, the constraints on that flow can be computed efficiently (on the order of a tenth of a millisecond) while supporting collaborative policy authorship. After introducing the core of language, this section discusses conflicts and conflict resolution, priorities, computational complexity, and illustrates the versatility of the language by explaining how to enforce NAT and VLAN-based policies.

FSL is based on nonrecursive DATALOG with negation, where each statement represents a simple if-then relationship. It relies on six keywords, five of which appear in the conclusions of rules: `allow`, `deny`, `waypoint`, `avoid`, and `ratelimit`. `allow` and `deny` each take eight arguments corresponding to the eight fields of a uniflow, and allow administrators to instruct NOX to allow and deny, respectively, any uniflow that matches a given set of conditions.

For example, the following four rules say that a superuser has no communication restrictions.

$$\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow superuser(U_s)$$
$$superuser(todd)$$
$$superuser(amy)$$
$$superuser(michelle)$$

The arguments to `allow` are all variables (denoted by symbols starting with a capital letter) and correspond to the user source, host source, access point source, user target, host target, access point target, protocol, and whether or not the flow is an initial request, respectively. The remaining three rules make simple declarative statements.

The keywords `waypoint`, `avoid`, and `ratelimit` take one extra argument in addition to the eight fields of a uniflow: the node that should be visited, the node that must not be visited, and the rate limit that should be imposed, respectively.

Administrators write DATALOG security policies using these five keywords to describe the NOX policy they would like enforced on uniflows. Every time a uniflow is initiated in an NOX network, the central controller queries the current FSL policy to determine which constraints should be applied. Each query supplies values for every one of the eight uniflow fields. If one of the fields is not known, because for example the uniflow concerns a machine not in the NOX network, the sixth keyword `unknown` occupies that field. Administrators can then write rules conditioned on unknown uniflow fields.

For example, to allow uniflows initiated by a desktop in the network to machines outside the network, an administrator could write the following rule.

$$\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow desktop(H_s) \wedge H_t = \texttt{unknown}$$

In addition to writing self-contained DATALOG policies, FSL allows administrators to write policies that reference external information sources. Those sources may be defined by, for example, SQL queries over local or remote databases, hash tables, or arbitrary procedural code. The definitions for those external sources accompany an FSL policy; we will not discuss them further. In examples, we will indicate external references by underlining the reference.

The formal syntax for FSL rules and FSL policies is given below. We use standard terminology. A predicate is a symbol with an associated arity. A term is a variable (starting with an upper-case letter) or an object constant (starting with a lower-case letter). An atom is a predicate of arity n applied to n terms. A literal is an atom or $\neg$ applied to an atom. An expression is ground if it contains no variables.

Definition 3 (FSL Rule)**.** *An FSL rule in the context of external references G takes the following form.*

$$h \Leftarrow b_1 \wedge \cdots \wedge b_n \wedge \neg c_1 \wedge \cdots \wedge \neg c_m$$

- *h, the head, and every b_i and c_i, collectively called the body, are atoms.*
- *Every variable in the body must appear in the head.*

- *Predicates* `allow` *and* `deny` *have arity eight;* `waypoint`, `avoid`, *and* `ratelimit` *have arity nine;* `unknown` *is an object constant.*
- *When* `waypoint`, `avoid`, *or* `ratelimit` *appear in the head, the last argument is an object constant.*
- *h does not contain any predicate $g \in G$.*

For those interested in technical details, notice the second condition: that every variable that appears in the body of a rule must appear in the head. This condition differs from the traditional safety[11] restriction in DATALOG in order to significantly simplify FSL's implementation. See Section 3.2.5 for details.

Definition 4 (FSL Policy)**.** *An FSL policy is a set of FSL rules Δ such that the dependency graph $\langle V, E \rangle$ is acyclic.*

V**:** *the set of predicates occurring in Δ*

E**:** *contains a directed edge from u to v exactly when there is a rule in Δ where the predicate in the head is u and the predicate v appears in the body.*

The semantics of FSL begins with the external references. From a logical perspective, every external reference implementation must ensure that every statement is either true or false—it must represent a model. The actual implementation could be a database or procedural code. Moreover, we assume the external references are aware of the reserved word `unknown`.

Definition 5 (External Reference Implementation)**.** *An implementation for the external reference g_i of arity n is a set of ground atoms of the form $g_i(a_1, \ldots, a_n)$.*

Entailment for FSL is based on the usual stratified semantics from deductive databases and logic programming [51].

Definition 6 (Basic Entailment)**.** *Let Δ be an FSL policy, Γ the external reference implementations, and ϕ a first-order sentence. Let M be the model that the stratified semantics assigns to $\Delta \cup \Gamma$ when the universe consists of the objects mentioned in Δ, Γ, and ϕ. $\models_M$ denotes the usual definition of satisfaction in the model M.*

$$\Delta \cup \Gamma \models \phi \text{ if and only if } \models_M \phi$$

One important feature of FSL is that the order in which rules appear is irrelevant. This makes combining a set of policies straightforward: collect the statements made in all of the policies. In contrast, languages where the order of statements is relevant, e.g. firewall configuration languages [57], make it hard for a computer system to combine independently authored policies automatically. Besides affording the independence administrators are accustomed to today when defining access control policies, embracing collaborative policy authoring has an additional benefit: real-world disagreements can be automatically pinpointed by policy analysis tools.

In FSL, a conflict arises for a given uniflow when a policy makes mutually exclusive decisions about what to do with that uniflow. Unlike traditional access control where all conflicts arise from being both granted and denied access, conflicts in FSL are produced in several different ways. Clearly a uniflow cannot be allowed and denied simultaneously. More interestingly, a uniflow cannot be denied and at the same time forced through an intermediate waypoint. Neither can a flow pass through a waypoint and avoid passing through that same waypoint.

Because NOX cannot enforce a conflicting policy (no uniflow can be both allowed and denied), conflict resolution is built into the language. Since FSL is a language concerned with security, its conflict resolution mechanism errs on the side of caution. A conflicting set of constraints is always resolved to produce the most restrictive constraint set. Deny is more restrictive than a set of required and prohibited waypoints,

[11] All variables must occur in a positive literal in the body.

and waypoints are more restrictive than a simple allow. A set of contradictory waypoints is equivalent to deny. A lower rate limit is more restrictive than a higher rate limit.

Similarly, because NOX cannot enforce an incomplete policy (every uniflow must be either allowed or denied), policy completion is also built into the language, but can easily be overridden. In contrast to conflict resolution, policy completion errs on the side of permissiveness by allowing all unconstrained uniflows. The rationale is a practical one: when installing a new system, it is useful to have all machines able to communicate to ensure the installation is working correctly.

The conflict resolution and policy completion schemes are built into the version of entailment that NOX uses to make authorization decisions. It relies on Basic Entailment (Definition 6) and ensures that every FSL policy corresponds to an NOX Policy (Definition 2).

Definition 7 (FSL Entailment)**.** *Suppose Δ is an FSL policy. FSL Entailment, $\models_{FSL}$ is defined in terms of Basic Entailment, $\models$.*

1. $\Delta \models_{FSL} \texttt{deny}(a_1, \ldots, a_8)$, *if*
   $\Delta \models \texttt{deny}(a_1, \ldots, a_8)$ *or* $\Delta \models \exists x.(\texttt{waypoint}(a_1, \ldots, a_8, x) \wedge \texttt{avoid}(a_1, \ldots, a_8, x))$.
2. $\Delta \models_{FSL} \texttt{waypoint}(a_1, \ldots, a_8, a_9)$, *if* $\Delta \models \texttt{waypoint}(a_1, \ldots, a_8, a_9)$ *and (1) does not hold.*
3. $\Delta \models_{FSL} \texttt{avoid}(a_1, \ldots, a_8, a_9)$, *if* $\Delta \models \texttt{avoid}(a_1, \ldots, a_8, a_9)$ *and (1) does not hold.*
4. $\Delta \models_{FSL} \texttt{allow}(a_1, \ldots, a_8)$, *if (1-3) do not hold.*
5. $\Delta \models_{FSL} \texttt{ratelimit}(a_1, \ldots, a_8, m)$ *if and only if m is the minimum of*
   $$\{maxrate\} \cup \{r | \Delta \models \texttt{ratelimit}(a_1, \ldots, a_8, r)\}$$

One of the benefits of building a conflict resolution mechanism into the language is that users can leverage it to express certain policies. For example, an open policy allows everything not explicitly denied. The following two rules deny all communication initiated by a blacklisted user but allow everything else.

$$\begin{array}{l} \texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \\ \texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \underline{blacklist}(U_s) \end{array} \tag{3.1}$$

In contrast, the conflict resolution mechanism does not make it easy to express a closed policy: where everything not explicitly allowed is denied. Adding permissive statements will not affect a too-restrictive policy; consequently, FSL provides a different mechanism for relaxing security. An FSL cascade, named for the cascading style sheets popular on the web, is a prioritized series of FSL policies, written $P_1 < \cdots < P_n$. Any policy in the ordering overrides all of the policies less than it.

For example, to define a closed policy, one could construct a cascade with two policies: $P_1 < P_2$. P_2 would describe all of the uniflows that should be allowed, and P_1 would contain a single rule:

$$\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req).$$

Definition 8 (FSL Cascade)**.** *An FSL cascade consists of a finite set of FSL policies $\{P_1, \ldots, P_n\}$ and a total ordering $<$ over those policies. We denote a cascade with $P_1 < \cdots < P_n$.*

In a cascade $P_1 < \cdots < P_n$, the highest ranked policy that says anything about a particular uniflow is the only policy that says anything about that uniflow. In other words, policy P_i determines the constraints on a uniflow if P_i constrains the uniflow and there is no other P_j that constrains that uniflow where $j > i$. The formal semantics is based on a precise definition of the constraints imposed on a uniflow by a policy.

Definition 9 (Uniflow Constraints)**.** *Consider a uniflow $u = \langle a_1, \ldots, a_8 \rangle$. When P is an FSL policy, let $C_P(u)$ be the smallest set that includes*

$$\begin{aligned}
\texttt{allow}(a_1, \ldots, a_8) \quad &\textit{if} \quad P \models \texttt{allow}(a_1, \ldots, a_8) \\
\texttt{deny}(a_1, \ldots, a_8) \quad &\textit{if} \quad P \models \texttt{deny}(a_1, \ldots, a_8) \\
\texttt{waypoint}(a_1, \ldots, a_8, a) \quad &\textit{if} \quad P \models \texttt{waypoint}(a_1, \ldots, a_8, a) \\
\texttt{avoid}(a_1, \ldots, a_8, a) \quad &\textit{if} \quad P \models \texttt{avoid}(a_1, \ldots, a_8, a) \\
\texttt{ratelimit}(a_1, \ldots, a_8, a) \quad &\textit{if} \quad P \models \texttt{ratelimit}(a_1, \ldots, a_8, a).
\end{aligned}$$

$C_P(u)$ denotes the set of constraints imposed by P on uniflow u.

Above we used $\models$, i.e. entailment without conflict resolution, resulting in a definition for Uniflow Constraints that allows conflicts. Replacing $\models$ with $\models_{FSL}$ produces a definition where conflicts have been resolved. The definition we provide loses less information than the alternative, and since resolving conflicts and computing the constraints imposed by a cascade are commutative operations (applying them in both orders produces the same result), the alternate definition can be given in terms of the one above.

Definition 10 (FSL Cascade Semantics)**.** *Consider a uniflow $u = \langle a_1, \ldots, a_8 \rangle$ and a cascade $P_1 < \cdots < P_n$. If P_u is the maximum P_i such that $C_{P_u}(u)$ is nonempty, then the constraints imposed on u by the cascade are exactly the constraints imposed on u by P_u.*

$$C_{P_1 < \cdots < P_n}(u) = C_{P_u}(u)$$

FSL was designed to make *writing* authorization policies convenient, but it was also designed to make *implementing* such policies efficient; hence, the complexity of FSL is important. The computational complexity of Basic Entailment turns out to be PSPACE-complete, just like traditional nonrecursive DATALOG with negation. Adding in conflict resolution and cascades increases the complexity by only a linear factor.

Theorem 1. *Let Δ be an FSL policy, Γ the external reference implementations, and ϕ a first-order sentence. Determining whether or not $\Delta \cup \Gamma \models \phi$ is PSPACE-complete.*

Proof. The inclusion in PSPACE follows because an FSL policy is a special case of a nonrecursive logic program with negation and without function constants, which is well-known to belong to PSPACE. The hardness proof demonstrates how to encode an arbitrary quantified boolean formula (QBF) as an FSL policy. The key insight into the proof is that existential variables can be simulated in the body of FSL rules by duplicating the rule for each possible variable instantiation. For example, the DATALOG rule $q(X) \Leftarrow p(X, Y)$, where Y can only be either a or b can be written in FSL as the following two rules.

$$\begin{aligned}
q(X) &\Leftarrow p(X, a) \\
q(X) &\Leftarrow p(X, b)
\end{aligned}$$

For encoding QBF formulae, this rule duplication technique is only polynomially larger than the usual DATALOG encoding. See Appendix A.2 for full details. □

The PSPACE complexity of FSL appears to destroy any hope of answering queries in the requisite tenth of a millisecond, but fortunately another of the usual DATALOG complexity results holds for FSL policies: restricting the arity of predicates to a constant reduces the complexity of entailment to polynomial time. The implementation discussed in Section 3.2.4 employs this constant-arity restriction to guarantee polynomial-time access control decision-making.

Finally, we illustrate how to express two standard security mechanisms for networks in FSL: network address translation (NAT) and virtual networks (VLANs). NAT provides security by disabling incoming connections for certain hosts, e.g. laptops, and disabling outgoing connections for servers.

$$\begin{aligned}
\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) &\Leftarrow \underline{laptop}(H_t) \wedge Req = true \\
\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) &\Leftarrow \underline{server}(H_s) \wedge Req = true
\end{aligned}$$

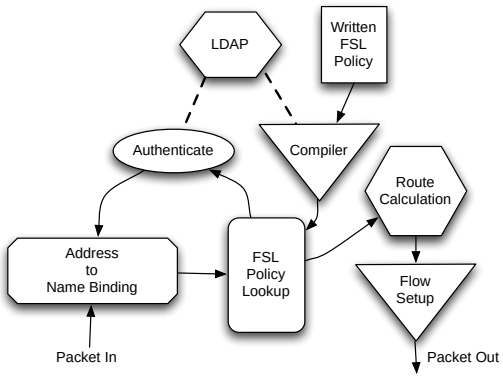

Figure 3.4: Component diagram of how FSL's implementation interacts with other NOX networking functions

Virtual networks give the appearance that certain hosts are isolated from the rest of the network. For example, the FSL implementation for a VLAN comprised of hosts a, b, and c is shown below.

$$vlan(a)$$
$$vlan(b)$$
$$vlan(c)$$
$$\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow vlan(H_s) \wedge \neg vlan(H_t)$$
$$\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \neg vlan(H_s) \wedge vlan(H_t)$$

3.2.4 Implementation

We have implemented FSL as the default policy language for NOX. Our implementation supports integration with external authentication stores (LDAP and AD), dynamic, incremental policy updates, and runtime group membership changes. All language features are supported with the exception that predicates (except the keywords) are restricted to one argument. Speed critical operations (such as runtime policy checking) are implemented in C++, while compilation and various integration components are written in Python.

In this section we describe our implementation and how it interoperates with other networking functions in NOX. We then describe our use of FSL internally. Finally, to demonstrate how the system performs under load (we are targeting networks of tens of thousands of hosts), we present performance analysis over stress-level workloads.

Operational Overview

Figure 3.4 shows how our FSL implementation integrates with NOX's standard network functions. It relies on other NOX applications to perform topology discovery, routing, authentication, and flow setup.

The FSL policy is declared in one or more files which are compiled into a low-level lookup tree. The compilation process checks all available authentication stores to verify that the principal names used in the policy file exist.

Packets received by NOX (for which there is no existing switch entry), are first tagged with all associated names and groups. The binding information between names and addresses happens at principal authentication. If binding information does not exist for the packet, the host and user are assumed to be unauthenticated, and the keyword `unknown` is used in place of the principal names.

In our implementation, we use an extension of FSL to control authentication policies (admission control). This is done by using the `unknown` keyword in FSL rules, and passing all matching packets to the authentication subsystem. This allows the administrator to specify the required authentication scheme as an action of an FSL condition. We describe this further in Section 3.2.4.

NOX implements FSL cascades using priorities. Each rule in a single cascade file is associated with the same priority level, with other files utilizing either lower or higher levels depending on their position in the hierarchy. For authenticated packets, the FSL subsystem finds the highest priority matching rule(s) and returns the resulting policy decision. If the decision is not a deny, the constraints are passed to the routing subsystem which will attempt to find a policy compliant route. Finally, NOX will set up the path in the network (specifying a rate limit if necessary) and re-inject the packet at the first hop switch. Once the flow is set up, all subsequent packets from the flow will be forwarded directly by the switch without having to go to the controller.

Compiler

The FSL compiler performs three tasks: it checks for valid usage of principals and predicates, it detects static conflicts, and it translates rules into a low-level byte format for insertion into a decision tree allowing fast lookup per packet.

Our compiler first parses the policy file and verifies that the principal names used exist in at least one of the available authentication stores. It also finds and reports detectable static rule conflicts; support for dynamic group membership obviates its ability to determine all possible conflicts. Conflicts arising due group membership changes are handled at runtime in the manner described in Section 3.2.3.

Once the file is parsed, the compiler stores the rules persistently in a canonicalized internal format. This is used to determine which rules have been added, removed or modified during a policy update, allowing for incremental updates of the policy at runtime. Generally, updates to the policy only require the addition and deletion of a few rules, rather than deleting and re-inserting the full policy.

Name to Address Bindings

NOX associates high-level names and groups with packet header values at authentication time. When a principle authenticates, its name is bound to the source access point, source MAC address, and source IP address (if it exists), used during the authentication exchange. The names and user credentials are generally retrieved from a remote authentication store (in the case of standard directory services). On succesful authentication, the names are cached locally at the controller. On each new flow, the source and destination header fields are hashed and used to retrieve the cached name information which is subsequently passed to the FSL lookup tree.

In principle, changes to group membership should be reflected immediately in the internal cache established at authentication time. While this is the case for our built-in authentication store, our implementation currently does not support dynamic updates to group membership in external directories. In this case, modification to the group membership of an authenticated principal may not take effect until it reauthenticates.

Policy Lookup and Enforcement

The policy evaluation engine is built around a decision tree intended to minimize the number of rules that need be checked per flow. The tree partitions the rules based on the eight uniflow fields and the set of groups, resulting in a compact representation of the rule set in a ten-dimensional space. Negative literals are ignored by the indexer and evaluated at runtime.

For example, the rule

$$\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = alice \wedge g(H_s) \wedge h(A_s) \wedge \neg p(A_t)$$

constrains U_s, mentions the source groups g and h, and does not mention any target groups. This rule would belong to the node in the decision tree where $U_s = alice$ and the source is constrained by either g or h.

Each node in the decision tree has one child for each possible value for the dimension that node represents, e.g. a node representing U_s has one child for each value U_s is constrained to in the subtree's policy rules. In addition, because some of a subtree's rules may not constrain the dimension a node represents, e.g. $Prot$ in the rule above, each node includes an ANY child for such rules to be placed in. Each node in the decision

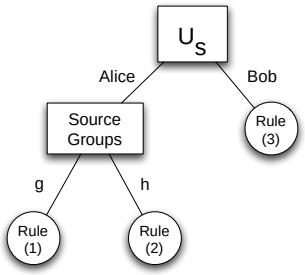

Figure 3.5: Example decision tree

tree is implemented using a hash table with chaining to ensure that each of its children can be found in near constant time. The decision as to which of the ten attributes to branch on at any point in the tree is made by finding the dimension that most widely segments a subtree's rule set. In particular, we select the dimension that minimizes the average number of rules at each child node plus the number of ANY rules in the subtree.

Recall that group membership is computed at authentication time. We will use G_s to denote all those groups to which the source of a uniflow belongs and G_t to denote the groups to which the target of a uniflow belongs. To find all rules that pertain to any given uniflow, the usual decision-tree algorithm is used with one exception: multiple branches may be followed at any given node. In particular, the ANY branch is always followed, and for branches splitting on source groups and target groups, all children that belong to the uniflow's G_s and G_t respectively, are followed.

For example, consider the policy that includes the following rules and the associated decision tree in Figure 3.5.

1. $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = alice \wedge g(H_s)$
2. $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = alice \wedge g(H_s) \wedge h(A_s) \wedge \neg p(A_t)$
3. $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = bob \wedge g(A_s)$

The root of the decision tree could either split on U_s or the source groups, since they are equally good at widely segmenting the rules. Suppose it splits on U_s. Then the root has two children: one for *alice* and one for *bob*. (Because every rule constrains U_s there is no need for ANY.) Rule (3) belongs to the *bob* child, which requires no further splitting. The other two rules are assigned to the *alice* child. The *alice* node is split on the source group dimension, where there are two options: g and h. Rule (1) is placed into the g child, and rule (2) is placed into either the g or h child. A uniflow originating from *alice* where the source fields only belong to group g would evaluate just rule (1), assuming (2) is placed into the h child and both (1) and (2) otherwise. If instead *alice* initiated a uniflow where the source fields belong to both g and h, then rules (1) and (2) are both evaluated, regardless of where rule (2) is placed in the tree.

The cost of answering an FSL entailment query is the cost of finding the pertinent rules in the tree index plus the cost of evaluating those rules. Rule evaluation is performed in the usual way. Conflict resolution is built into the system: actions of all matching rules are collected, and the most secure, potentially composite, action is returned. An FSL cascade is implemented by evaluating rules in priority order; once a match is found, only the remaining rules of the same priority are evaluated.

Bootstrapping Authentication

Authenticating users and hosts generally requires some default connectivity which is dependent on the admission policy. For example, we are preparing for a deployment in a large university in which all users with private addresses must authenticate via a captive web portal before being given access to the network. Therefore, we have to provide default access from hosts to web-servers, and from the web-server to the directory servers.

We have extended our FSL implementation to also support the declaration of admission control policies and the requisite connectivity for authentication. Similar to access controls, the admission policy is a set of FSL-like rules in which the action denotes the authentication type. The predicates used in the rules include access points (*e.g.*, $wireless(A_s)$), protocols, IP prefixes, and special purpose built-in groups such as *all-registered-macs*. For example, the following policy snippet sets up default connectivity for authentication, and redirects all unauthenticated wired hosts to a captive web portal.

$$\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = ARP$$
$$\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = DHCP$$
$$\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_s = authentication\_server \wedge Prot = HTTP$$
$$\texttt{http-redirect}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_s = \texttt{unknown} \wedge Prot = HTTP$$

$$\texttt{cascade}()$$

$$\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_s = \texttt{unknown}$$

In our network, all hosts on wired access points are automatically authenticated if they have a registered MAC address. Whereas users connecting via wireless are required to use a captive web portal. While logically the admission control policy is distinct from the access control policy (since automated conflict resolution is no longer viable), we use the same internal mechanisms to enforce it.

Experience and Performance

The security policy of our local area network, consisting of roughly 50 hosts connected through 4 switches, is managed by our FSL implementation running on NOX. The policy contains 24 rules declared over 64 principals and 11 groups including workstations, servers, printers, and mobile devices.

We include a subset of the policy in Appendix A.1. Roughly, our internal policy allows unrestricted ARP and SSH between all hosts. Test servers are not allowed to communicate externally. Servers and printers should only allow inbound connections (except for outgoing SSH sessions on server) providing protection similar to a DMZ. Laptops and mobile devices (such as mobile phones supporting Wi-Fi) are not allowed inbound connections (similar to NAT). We also have a few rules which allow monitoring and diagnostic traffic between an administrative host and all switches. The lowest priority rule in the policy file matches all flows and denies the traffic.

Round trip flow setup latencies (involving two permission checks, route calculations, and flow-entry setups) are generally under 50ms. Once the policy decision has been made for each uniflow, all subsequent packets for that flow are forwarded at line speed (as supported by the switch forwarding hardware). On a network of this size, there is not enough traffic to stress our implementation (we generally see less than 20 new flow setups/s). However in benchmarks using generated traffic, our implementation running with our internal policy file supports permission checks on over 60,000 flows/s, an order of magnitude higher than any network we have measured. We present more performance tests of the system under load below.

Our experience has been that our simple rule set provides sufficient connectivity for day to day operations without requiring constant maintenance. Furthermore, we find that almost all of the rules are declared over classes of devices that connect to the network (as opposed to individual machines or users) and therefore we expect the policy file to grow slowly with the number of managed principals. We expect to explore this in more detail shortly as we are currently preparing for two much larger deployments of FSL in networks of hundreds and thousands of hosts.

In preparation for broader deployment we have tested our implementation's performance under generated workloads. Unfortunately it is difficult to produce meaningful results without access to sample policies written for real networks. In the degenerate case, enforcement of a policy with many ANY rules can scale linearly as the rule set size grows. Conversely, a large policy containing only exact match rules could force construction of a maximum depth tree, incurring significant hashing overhead.

We test the performance and memory overhead of our implementation with policy files of increasing size (table 3.1). All policies (except those with 0 rules) forced a maximum depth tree once constructed. For each

incoming flow, on average $log_2(\#rules)$ matched and had to be evaluated by the system. Table 3.2 shows the same test using policies in which 10% of the rules contain ANY fields (the number of fields containing ANYs is evenly distributed between 1 and the maximum number of fields). In this case, the increased number of rules matching a given flow due to the number of ANYs causes a performance degradation in larger policy files.

The goal of this analysis is not to provide an exhaustive investigation of the performance of our implementation, but rather to gain some insight into its handling of load under various rule sets. As shown in [21, 20], even large enterprise networks of tens of thousands of hosts generally have less than 10,000 flow requests per second, far below the performance capabilities of our implementation for the rule sets tested.

	flows/s	Mbytes	avg. matches
0 rules	103,699	0	0
100 rules	78,808	0	4
500 rules	78,534	1	6
1,000 rules	75,414	2	7
5,000 rules	71,702	9	8
10,000 rules	67,843	56	9

Table 3.1: Performance and memory overhead of our FSL implementation over policies with increasing rule count. All policies contain exact match rules and average $log_2(\#rules)$ matching rules per flow. The rightmost column contains the average number of matching rules per flow.

	flows/s	Mbytes	avg. matches
0 rules	103,699	0	0
100 rules	100,942	1	2
500 rules	85,373	1	4
1,000 rules	76,336	2	10
5,000 rules	54,416	9	30
10,000 rules	46,956	38	52

Table 3.2: Performance and memory overhead of our FSL implementation over policies declared over 1000 principals in which 10% of the rules contain ANYs. The rightmost column contains the average number of matching rules per flow.

3.2.5 Related Work

FSL is based on a restricted form of DATALOG [36, 10, 41, 54, 28, 42, 9] with negation [48, 10, 36] so that authorization policies can be authored in distributed settings [13, 35, 48, 7]. Because disagreements naturally arise among administrators, FSL was designed so that those disagreements would manifest as conflicts in policies [53, 36, 12, 10, 24, 27, 48, 11, 7]. Conflicts between policy modules entered by different administrators can be detected by automated methods [44, 7, 36] that can be built into policy management tools. At the same time, because FSL policies are used by real systems [14, 26, 52], any conflicts that are not resolved by administrative tools must be resolved automatically at enforcement time [53, 36, 12, 10, 24, 27, 48, 11]. Unlike prior languages, FSL conflict resolution is complicated by the fact that policies can reference external sources [48, 13, 41], which can change independent of the policy. This feature of FSL is essential for large-scale deployment, since it is impractical to ask large organizations to rewrite their organization charts in a network policy language. Finally, FSL employs priorities (called "overrides" in [13] and "exceptions" in [12]) via FSL Cascades, to help administrators express certain types of policies.

FSL lacks certain features found in the literature. The most common reason a feature was excluded is the performance requirement. Below, we enumerate some of the limitations of FSL and explain why the limitations exist.

Datalog based. Other languages are based on fundamentally different and often more expressive logics [32, 53, 5, 26, 27, 8, 24]. First, DATALOG was chosen because of implementation concerns: the simpler the language, the simpler and faster the implementation. Second, additional expressiveness allows users to state policies that can be difficult to enforce. In first-order logic, a policy might say to allow one of two uniflows without saying which. Third, DATALOG is closer to traditional CS programming languages than other logics, which is important because policy authors are network operators (who may be logic novices).

Existential variables are disallowed. Existential variables (those that occur in the body of a rule but not in the head) are usually included in DATALOG languages because they are useful in a variety of circumstances. For example, Role-based Access Control (RBAC) has been explained very concisely using existential variables [25]. Without existential variables, the implementation is far simpler because there is no search through variable assignments. $q(X) \Leftarrow p(X,Y) \wedge r(Y)$ says that $q(t)$ is true if there is some u

such that $p(t, u) \wedge r(u)$ is true. The implementation needs to search through the possible assignments to Y. FSL's design and application ensure that such a search need never occur.

Recursion is disallowed. Policy languages that support recursion without negation [41], stratified recursion [36] and arbitrary [10] recursion occur in the literature. Without recursion, FSL's implementation is both simpler and faster.

Conflict resolution is built into the language. Sometimes the conflict-resolution scheme for a language is built-in [48, 24, 11], but sometimes it is defined by the user [10, 53, 27]. Sometimes, conflicts are only detected but not resolved [7], and other times conflicts are handled by a combination of detection and user-defined resolution [36]. Our choice to fix the conflict resolution scheme comes about because conflicts and conflict resolution in FSL are more complex than usual.

Unlike common access control decisions, which either allow or deny an action, each decision in FSL consists of a set of constraints, e.g. pass through waypoint A but do not pass through B. Conflict resolution consists of mapping one set of constraints to another set; moreover, only certain combinations of constraints are actually enforceable. A system cannot force a uniflow through waypoint A and at the same time avoid waypoint A. It is likely that if conflict-resolution were left to user specification, some conflicts would not actually be resolved, and a built-in resolution mechanism would be necessary anyway.

Delegation is not directly supported. One common concern in authorization logics is the ease with which one principal can delegate rights to another [26, 43]. Such concerns are important when there are certain actions that can change which future actions are allowed and denied. In the setting for which FSL was designed, there are no rights-changing actions. The answer to all authorization requests is the same for all time (except for external reference changes); hence, delegation primitives have been omitted.

FSL policies are timeless. Some policy languages include constructs that reference temporal events [34, 26, 53, 43, 48, 39], e.g. if A communicated with B in the past, then disallow B from communicating with C. History acrues very quickly with 10^4 flows initiated per second. Storing that much information is impractical, and even if it were practical, the system would not be able to answer queries about it quickly enough.

Policy enforcement is centralized. Languages built for trust management are designed so that the authorization policy can be authored by a group of people working independently. But unlike FSL, which enforces that policy at a central location, trust management languages enforce each policy in a distributed fashion [5, 39, 54, 42, 40, 49, 9]. Distributed enforcement is the subject of future work.

Metalevel policy operations are limited. FSL supports two metalevel policy operations: combining a set of policies, and prioritizing a set of policies (FSL cascades). Among the other metalevel operations included in [13], the most germane is scoping, also used in [48]. Scoping is an operation that restricts a given policy to a certain class of objects. For example, in a university setting scoping could restrict a CS administrator from constraining interdepartmental server communications in the Math department. Adding scoping to FSL is the subject of future work.

Finally, it is worth highlighting the features of FSL that we have been unable to find elsewhere in the literature.

Access control decisions are constraint sets. Instead of either allowing or denying every request, FSL prescribes a set of constraints that apply to that request. Allow and deny are special cases.

Conflict resolution maps one constraint set to another. Because access control decisions result in a set of constraints, a conflict can be attributed to three or more of those constraints. Conflict resolution takes as input one constraint set and outputs another constraint set.

3.2.6 Discussion

Motivated by the opportunities provided by emerging network technologies, and the need for network operators and administrators to express general constraints using distributed and incrementally authored policies enforced efficiently at line speeds, we developed a network security policy framework around a language expressing flow-based network policies. Our policy language FSL allows operators to selectively allow network access, while also imposing constraints on usage rate, allowed network paths (which may be required to avoid or include certain nodes), and directionality (such as restricting clients to outbound flows only). As

the name *Flow Security Language (FSL)* implies, the key underlying concept is network flow. In particular, after trying several alternatives, we decided to take unidirectional flows as the basic unit of network access, regarding bidirectional flow as comprising two related uniflows. By associating additional characteristics such as flowrate and initiating party with each flow, we achieve a systematic network flow analog of the traditional access-control matrix (of, say, users, filenames, and read-write-execute permissions). Building on this foundation, FSL allows succinct, structured, high-level specification of allowed flows, freeing network administrators from the drudgery of configuring myriad router ACLs, firewalls, NATs and VLANs to achieve comprehensive and conceptually straightforward network usage policies.

FSL is a declarative policy language based on nonrecursive DATALOG with structured negation. The declarative nature of FSL makes it possible for separate network administrators to separately express their policies and automatically combine those policies. We show how combined policies with potentially conflicting components can be enforced efficiently. In addition, logic-based algorithms that detect and resolve conflicts in predictable and reliable ways may be incorporated into policy development environments. FSL also supports prioritized policy combination, which is a natural way to express many policies and enables incremental policy updates.

We demonstrated FSL's use in practice by implementing it within the NOX network architecture, running it in our internal network for nearly a year, and subjecting the system to additional tests using much more demanding artificially generated loads. We show through performance analysis that our implementation has modest memory requirements and can scale to very large networks while supporting policy files of tens of thousands of rules. We are preparing to further explore FSL's application in operational neworks through much larger deployments in the near future.

Chapter 4

Results and Discussion

The previous chapter covered the detailed technical results on each of the two topic areas, NOX and FSL. In this discussion, we step back and discuss the more general lessons we learned from building and using our system.

The largest lesson we learned is that (once deployed) we found it much easier to manage the network than we expected. On numerous occasions we needed to add new switches, new users, support new protocols, and prevent certain connectivity. On each occasion we found it natural and fast to accomplish these tasks. There is great peace of mind to knowing that the policy is implemented at the place of entry and determines the route that packets take (rather than being distributed as a set of filters without knowing the paths that packets follow). By journaling all registrations and bindings, we were able to identify numerous network problems, errant machines, and malicious flows—and associate them with an end-host or user. This bodes well for network managers who want to hold users accountable for their traffic or perform network audits.

We have also found it straightforward to add new features to the network: either by extending the policy language, adding new routing algorithms (such as supporting redundant disjoint paths), or introducing new application proxies as waypoints. Overall, we believe that our approach's most significant advantage comes from the ease of innovation and evolution. By keeping the switches dumb and simple, and by allowing new features to be added in controller software, rapid improvements are possible.

Chapter 5

Conclusions

As in all software systems research, "the proof is in the using of the system" and at this point we have substantial experience with it. We have been running it internally throughout this year's project, and we now have trial deployments at several universities. Our basic conclusion is that the system:

- **Makes network management easier:** by providing a general programmatic interface, with a full set of name-address bindings, network operators can accomplish complex tasks with a minimum of effort.
- **Makes networks more secure:** our approach both secures networks at a low-level (*i.e.*, preventing ARP-spoofing) and also provides the ability to control access in a flexible manner.
- **Makes network hardware less expensive:** by taking the intelligence out of switches, one can use relatively inexpensive commodity switches.

Chapter 6

References

[1] Alterpoint homepage. `http://www.alterpoint.com`.

[2] Consentry networks homepage. `http://www.consentry.com`.

[3] Nevis networks homepage. `http://www.nevisnetworks.com`.

[4] OpenFlow Switch Consortium homepage. http://www.openflowswitch.org/.

[5] M. Abadi, M. Burrows, and B. Lampson. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[6] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *HotNets-VI*, 2007.

[7] A. Barth, J. C. Mitchell, and J. Rosenstein. Conflict and combination in privacy policy languages. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society*, 2004.

[8] D. Basin, E.-R. Olderog, and P. E. Sevinc. Specifying and analyzing security automata using CSP-OZ. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 70–81, 2007.

[9] M. Y. Becker, C. Y. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.

[10] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.

[11] E. Bertino, E. Ferrari, F. Buccafurri, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proceedings of the IEEE Computer Security Foundations Workshop*, 1999.

[12] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2):101–140, 1999.

[13] P. A. Bonatti, S. D. di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 164–173, 2000.

[14] K. Borders, X. Zhao, and A. Prakash. CPOL: High-performance policy evaluation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 147–157, 2005.

[15] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI '05*, 2005.

[16] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. S. E. Ng. Maestro: A Clean-Slate System for Orchestrating Network Control Components. preprint, 2008.

[17] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting edge of ip router configuration. *SIGCOMM Computer Comm. Rev.*, vol. 34, no. 1, pp. 21–26, 2004.

[18] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting edge of ip router configuration. *SIGCOMM Computer Comm. Rev.*, 2004.

[19] K. L. Calvert, W. K. Edwards, and R. E. Grinter. Moving Toward the Middle: The Case Against the End-to-End Argument in Home Networking. In *HotNets-VI*, 2007.

[20] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM Conference*, Kyoto, Japan, Aug. 2007.

[21] M. Casado, T. Garfinkle, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. 15th USENIX Security Symposium*, Vancouver, BC, Aug. 2006.

[22] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM '07*, 2007.

[23] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Usenix Security Symposium*, 2006.

[24] L. Cholvy and F. Cuppens. Analyzing consistency of security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[25] J. Crampton. Understanding and developing role-based administrative models. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 158–167, 2005.

[26] J. Crampton, G. Loizou, and G. Oshea. A logic of access control. *The Computer Journal*, 44(1):137–149, 2001.

[27] F. Cuppens, L. Cholvy, C. Saurel, and J. Carrere. Merging security policies: analysis of a practical example. In *Proceedings of the IEEE Computer Security Foundations Workshop*, 1998.

[28] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[29] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. In *ACM SIGCOMM Computer Communication Review*, 2005.

[30] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards and operating system for networks. In *ACM SIGCOMM Computer Communication Review*, July 2008.

[31] C. Gunaratne, K. Christensen, S. Suen, and B. Nordman. Reducing the Energy Consumption of Ethernet with an Adaptive Link Rate (ALR). *IEEE Transactions on Computers*, forthcoming.

[32] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2003.

[33] B. Heller and N. McKeown. A comprehensive power management architecture. Work in progress, 2008.

[34] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 134–143, 2006.

[35] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[36] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42. IEEE Press, 1997.

[37] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, 2004.

[38] Z. Kerravala. Configuration management delivers business resiliency. *The Yankee Group*, Nov. 2002.

[39] A. J. Lee and M. Winslett. Safety and consistency in policy-based authorization systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 124–133, 2006.

[40] N. Li, B. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. In *Proceedings of the ACM Transactions on Information and System Security*, pages 128–171, 2003.

[41] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Symposium on Practical Aspects of Declarative Languages*, 2003.

[42] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[43] N. Li and M. V. Tripunitara. On safety in discretionary access control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 96–109, 2005.

[44] N. Li, M. V. Tripunitara, and Q. Wang. Resiliency policies in access control. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 113–123, 2006.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[46] P. Newman, G. Minshall, and T. L. Lyon. IP switching - ATM under IP. *IEEE/ACM Trans. Netw.*, 6(2):117–129, 1998.

[47] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-Wide Decision Making: Toward A Wafer-Thin Control Plane. In *HotNets III*, 2004.

[48] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium*, 2001.

[49] R. L. Rivest and B. Lampson. SDSI - a simple distributed security infrastructure. Technical report, Massachusetts Institute of Technology, 1996.

[50] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: Enabling controlled networking. *SIGCOMM Computer Comm. Rev.*, vol. 33, no. 1, 2003.

[51] J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[52] D. S. Wallach and E. W. Felton. Understanding java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, 1998.

[53] D. Wijesekera and S. Jajodia. Policy algebras for access control - the predicate case. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 171–180, 2001.

[54] M. Winslett, C. C. Zhang, and P. A. Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 168–179, 2005.

[55] A. Wool. The use and usability of direction-based filtering in firewalls. *Computers & Security*, vol. 26, no. 6, pp. 459–468, 2004.

[56] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[57] L. Yuan and H. Chen. FIREMAN: A toolkit for firewall modeling and analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 199–213, 2006.

[58] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, and G. Hjalmtysson. Routing design in operational networks: A look from the inside. *Proc. SIGCOMM*, Sept. 2004.

Appendix A

Appendices

A.1 Example Policy Cascade

The following policies are ordered from highest to lowest: $P_1 < P_2 < P_3 < P_4$. The first policy shown below overrides all those after it—likewise for the other policies.

Policy P_4
 # allow ARP and DHCP
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = arp$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = dhcps \wedge H_t = gateway$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = ssh \wedge \underline{computer}(H_s)$

 # allow computers to ssh anywhere
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = ssh \wedge \underline{computer}(H_s)$

 # allow internal monitoring flows: proprietary protocols registered with the system
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1616 \wedge H_s = badwater$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1717 \wedge H_s = badwater$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1818 \wedge H_s = badwater$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1616 \wedge H_t = badwater$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1717 \wedge H_t = badwater$
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1818 \wedge H_t = badwater$

Policy P_3
 # dissallow testing machines from communicating externally
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow testing(H_s)$
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow testing(H_t)$

 # servers should be inbound-only
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{server}(H_s)$
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{printer}(H_s)$

 # laptops and mobile devices should be outbound-only
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{mobile}(H_t)$
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{laptop}(H_t)$

Policy P_2
 # allow known devices to communicate as long as they abide by the
 # previous rules.
 $\texttt{allow}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow all(H_s)$

Policy P_1
 # default deny
 $\texttt{deny}(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req)$

A.2 Proofs

Theorem 2 (Entailment for FSL is PSPACE-complete)**.** *Given an FSL rule set Δ and an atom a, checking whether $\Delta \models a$ is PSPACE-complete.*

Proof. Because FSL is a form of function-free, recursion-free logic programming, and entailment for the latter is PSPACE-complete, entailment for FSL belongs to PSPACE. To show entailment is PSPACE-hard, we perform a reduction from QBF.

Consider any quantified boolean formula in prenex form: $Q_1 x_1 \dots Q_n x_n . \phi(x_1, \dots, x_n)$. The FSL query that represents the value of this QBF will be written as

$$val_{Q_1 x_1 \dots Q_n x_n . \phi(x_1, \dots, x_n)}.$$

The proof demonstrates how to define this predicate in polynomial time. The first step is quantifier elimination.

If Q_1 is $\forall$, then the query is defined as follows.

$$val_{Q_1 x_1 \dots Q_n x_n . \phi(x_1, \dots, x_n)} \Leftarrow val_{Q_2 x_2 \dots Q_n x_n . \phi(x_1, \dots, x_n)}(1) \wedge val_{Q_2 x_2 \dots Q_n x_n . \phi(x_1, \dots, x_n)}(0)$$

Here 1 represents true, 0 represents false, and $val_{Q_2 x_2 \dots Q_n x_n . \phi(x_1, \dots, x_n)}$ is a unary predicate that is true for all those u such that $Q_2 x_2 \dots Q_n x_n . \phi(u, x_2, \dots, x_n)$ is true. Otherwise, Q_1 is $\exists$, and the definition is given by two rules.

$$val_{Q_1 x_1 \dots Q_n x_n . \phi(x_1, \dots, x_n)} \Leftarrow val_{Q_2 x_2 \dots Q_n x_n . \phi(x_1, \dots, x_n)}(1)$$
$$val_{Q_1 x_1 \dots Q_n x_n . \phi(x_1, \dots, x_n)} \Leftarrow val_{Q_2 x_2 \dots Q_n x_n . \phi(x_1, \dots, x_n)}(0)$$

Both cases require constructing definitions for additional predicates. The process is very similar to what is shown above. After constructing k definitions, we need to define

$$val_{Q_{k+1} x_{k+1} \dots Q_n x_n . \phi(x_1, \dots, x_n)}(x_1, \dots, x_k).$$

If Q_{k+1} is $\forall$, the definition is as follows; otherwise, the definition requires two rules as shown above.

$$val_{Q_{k+1} x_{k+1} \dots Q_n x_n . \phi(x_1, \dots, x_k)}(x_1, \dots, x_k) \Leftarrow$$
$$val_{Q_{k+2} x_{k+2} \dots Q_n x_n . \phi(x_1, \dots, x_n)}(x_1, \dots, x_k, 1) \wedge val_{Q_{k+2} x_{k+2} \dots Q_n x_n . \phi(x_1, \dots, x_n)}(x_1, \dots, x_k, 0).$$

Notice two things. First, in every rule, the variables in the body also appear in the head, making each rule a valid FSL rule. Second constructing the rules corresponding to each quantifier takes time linear in the original sentence, and there are no more than a linear number of quantifiers; thus, the total cost is at most quadratic in the size of the original sentence.

After eliminating all n quantifiers, the rule set requires a definition for

$$val_{\phi(x_1, \dots, x_n)}(x_1, \dots, x_n)$$

where $\phi(x_1, \dots, x_n)$ is quantifier-free. The definition should be true for exactly those assignments to x_1,...,x_n that satisfy the boolean formula $\phi(x_1, \dots, x_n)$. Suppose that $\phi(x_1, \dots, x_n)$ is written in conjunctive normal form.

$$((p_{11} \vee \dots \vee p_{1k_1}) \wedge \dots \wedge (p_{m1} \vee \dots \vee p_{mk_m}))$$

Here p_{ij} represents x_l or $\neg x_l$ for l in $\{1, \dots, n\}$.

Just like the case of quantifiers, the definition for $val_{\phi(x_1, \dots, x_n)}(x_1, \dots, x_n)$ is broken into a definitions for a series of predicates: val_1,...,val_m. Predicate val_i corresponds to the evaluation of $\phi(x_1, \dots, x_n)$ when only considering the first i clauses. The definitions are indicated below. We use two macros. $var(p_{ij})$ is replaced by the variable mentioned in p_{ij}, and $sign(p_{ij})$ is replaced by 1 if the literal p_{ij} is positive and 0 otherwise. Thus, each instance of $var(p_{ij}) = sign(p_{mn})$ is positive equality atom comparing a variable and 1 or 0, e.g. $x = 1$.

$$
\begin{aligned}
&val_1(x_1, \ldots, x_n) \Leftarrow var(p_{11}) = sign(p_{11}) \\
&\vdots \\
&val_1(x_1, \ldots, x_n) \Leftarrow var(p_{1k_1}) = sign(p_{1k_1}) \\
&val_2(x_1, \ldots, x_n) \Leftarrow var(p_{21}) = sign(p_{21}) \wedge val_1(x_1, \ldots, x_n) \\
&\vdots \\
&val_2(x_1, \ldots, x_n) \Leftarrow var(p_{2k_2}) = sign(p_{2k_2}) \wedge val_1(x_1, \ldots, x_n) \\
&\vdots \\
&val_m(x_1, \ldots, x_n) \Leftarrow var(p_{m1}) = sign(p_{m1}) \wedge val_{m-1}(x_1, \ldots, x_n) \\
&\vdots \\
&val_m(x_1, \ldots, x_n) \Leftarrow var(p_{mk_m}) = sign(p_{mk_m}) \wedge val_{m-1}(x_1, \ldots, x_n)
\end{aligned}
$$

Because $val_m(x_1, \ldots, x_n)$ corresponds to the truth value of $\phi(x_1, \ldots, x_n)$ when taking all m clauses into account, it is exactly the definition we need.

$$val_{\phi(x_1,\ldots,x_n)}(x_1, \ldots, x_n) \Leftarrow val_m(x_1, \ldots, x_n)$$

Again, every variable in the body also appears in the head. The number of rules is equal to the number of literals in conjunctive normal form, and each rule is linear in the number of variables. Thus, the resulting rules are polynomial in the size of the quantifier-free portion of the formula.

This completes the encoding. The time to produce the encoding is polynomial in the size of the original QBF formula, which completes the proof of PSPACE-hardness. □

Symbols, Abbreviations, Acronyms

- NOX: Network Operating System
- FSL: Flow-Based Security Language
- VLAN: Virtual Local Area Network
- OF: OpenFlow